(12)

UTEC-82-020

AD A112824

SEMIANNUAL TECHNICAL REPORT

TRANSFORMATION of ADA PROGRAMS INTO SILICON

81 Sept. 1 - 82 Feb. 28

Elliott I. Organick, Principal Investigator
(801) 581-6087

Contractor: The University of Utah

Date of Contract: 81 SEPT 1

Expiring: 83 AUG 31

Sponsored by

Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4305

Under Contract No. MDA 903-81-C-0411, issued by
Defense Supply Service - Washington, Washington DC 20310

March 1982

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: UTEC-82-020	2. GOVT ACCESSION NO.: AD-A112 824
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): TRANSFORMATION OF ADA PROGRAMS INTO SILICON	5. TYPE OF REPORT & PERIOD COVERED: semi-annual, 81 Sept 1 - 82 Feb 28
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Dr. E. Organick, Dr. G. Lindstrom, D. K. Smith, Dr. Subrahmanyam, T. Carter	8. CONTRACT OR GRANT NUMBER(s): MDA 903-81-C-0411, ARPA Order-4305
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Utah, Computer Science Department, Salt Lake City Utah 84112	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 1001/1122
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency (DoD), 1400 Wilson Boulevard, Washington, D.C. 22209	12. REPORT DATE: March 1982
	13. NUMBER OF PAGES:
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Defense Supply--Service Washington, Rm 1d-245, The Pentagon, Washington, D.C. 20310	15. SECURITY CLASS. (of this report): unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): control unit, CADDET, SPICE Internet protocol, submodules, Ada-to-silicon, transformation methodologies, high level program specifications, DoD Internet Protocol, special function architecture, ADA packages & tasks, VLSI synthesis, program formal specifications, device modeling, switched capacitor filter, stored logic array, logic simulator, hand shake, speed-independent, one-hot, portable standard LISP, silicon compiler, VLSI

This report outlines the beginning steps taken in an integrated research effort toward the development of a methodology, and supporting systems, for transforming Ada programs, or program units, (directly) into corresponding VLSI systems. The time seems right to expect good results. The need is evident; special purpose systems should be realistic alternatives where simplicity, speed, reliability, and security are dominant factors. Success in this research can lead to attractive options for embedded system applications.

cont

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada programs can be regarded as ensembles of machines, one per program unit (module), which in turn may be mapped directly into corresponding VLSI structures on one or more chips with interconnecting (packet switched or other) communication nets.

Many of the transformation steps, when performed manually, when optimization is not everywhere crucial, and when care is taken to constrain somewhat the structure of the source Ada program, appear to be understood.

The research reported here is part of a five-year plan, the first year of which focuses on "proving" the concepts through a realistic demonstration of methodology for a specific example Ada program (a silicon representation of part or all of the DoD Standard Internet Protocol, IP, initially expressed in Ada.). Since the mapping from Ada to VLSI is seen as a multistep, iterative procedure, considerable effort for the following four and a half years will be the invested in the development and tailoring of intermediate languages and their bridging algorithms (compilers), as needed, and in the development of objective criteria for their use with feedback loops for iterative design.

Implicit in these objectives is the development of a set of hardware structuring paradigms (rewrite rules) whose application can ensure that transformation steps between levels of abstraction in the design process are well structured in order to preserve the integrity and, where possible, the clarity of the original Ada specification. Some paradigms, but of course not all, lead to highly efficient implementations.

Table of Contents

List of Figures

Summary

This report summarizes the first six months work of the research project, "Transformation of Ada Programs into Silicon." Our project has five main objectives:

1. Develop and document elements of a transformation methodology for converting Ada programs, or program constructs, into VLSI systems which are ensembles of intercommunicating state machines. This research includes: selecting intermediate languages, as deemed necessary, and identifying a sufficient set of transformation rules for mapping program specifications through successive levels of representation, from Ada to integrated circuits.

2. Demonstrate the methodology developed in 1 by manually applying it to a non-trivial example: transforming an Ada-encoded representation of the DoD Standard Internet Protocol (or a significant subset thereof) into NMOS circuitry.

3. Work toward a theory for identifying substructures within Ada programs for which the transformation methodology is attractive or suitable, according to pragmatic considerations. (That is, attempt to determine the advantages and disadvantages of converting prototypical Ada constructs into silicon.) Identify and classify those Ada program constructs that are especially "good" candidates for conversion.

4. Develop specifications for a set of software tools for use in automating the transformation methodology developed in 1.

5. Develop a methodology for testing integrate circuits representing Ada program units and for integrating such circuits into a larger system embodying the remainder of the Ada program of which the produced integrated circuit was extracted.

Thus far, work has been pursued in four main areas:

1. Converting the DoD Internet Protocol to silicon.

This is the project's principal case study. Section 2 provides an overview

L. Bum Ac

of our work thus far in this area. We summarize here the key points to be found in that section:

- Major design decisions:

    1. The first is to split the INM into three submodules: an INM_OUT dealing with traffic outbound on a given local net, an INM_IN similarly handling inbound traffic, and an INM_SRV tying them together and interfacing to the Host(s). We envision one INM_IN and INM_OUT pair for each local net interface, but only one INM_SRV per INM.

    2. A systematic control paradigm has been developed for use at each submodule interface. This paradigm involves a two-phase Ada rendezvous, whereby the initiator of each data transfer performs an entry call on the corresponding receiver task. When the data transferred has been fully processed, a reciprocal rendezvous takes place to report the success or failure of that processing.

- Development and testing plan: A five-level software development and testing plan has been constructed, with the levels corresponding to IP applications in increasingly generalized settings. The plan stipulates testing as each level is reached, rather than as an epilog to the development plan.

- Schedule: This section concludes with some Year One and Year Two goals for the completion of IP case study software and hardware development. Our strategy is for software development to "lead" hardware development by one or more levels, so that confidence is increased on the finality of designs as they are committed to hardware.

2. Identifying Ada program structure candidates for transformation to silicon. These concepts and conclusions, reported in more detail in Section 3, are summarized as follows:

- Ada programming methodology not only implies a system software design methodology, but also can imply a hardware implementation methodology. and can serve as a guide for design automation.

- Languages like Ada, which offer data abstraction and concurrent tasks, form their own base for a design methodology. A small set of high-level, Ada-specified structuring rules, using Ada packages and tasks as building blocks, appears necessary for "steering" the transformation from a subensemble of an Ada program graph toward custom circuit equivalents.

- Ada-like programs provide a natural model-building paradigm which is: ensembles of (state machine, data path) pairs. Use of such engines can lead to reasonably high degrees of concurrency in many cases. This "high-level concurrency" is augmented with "low-level concurrency" achieved within the circuits of individual engines.

- A Special Function Architecture derived from an Ada program unit is always interchangeable with a compiled image of the original Ada code. That is, an SFA can be replaced by compiled machine code loaded onto a general-purpose interpreter.

- An SFA is therefore testable as a program unit, since both are expected to have the same semantics.

- Ada no doubt has limitations as a design vehicle, but seems appropriate for many applications. It is too early to tell whether and how Ada should be enriched for use over the entire spectrum of the transformation process. It is too early to tell precisely for what problem domain Ada is best suited as the starting specification language. It is even too early to tell if Ada programs, for many useful applications, are, as many suspect, over-specified. Section 5 of this report expands on these issues.

- Experience transforming non-toy examples from Ada to silicon, such as Utah's DoD-IP to NMOS project, should answer a number of our questions on the applicability of this methodology. Section 2 illustrates the application of our design philosophy and methodology to this important case study.

- Advances in transformation system techniques and tools appear to hold the key to the feasibility of fully or partially automating the conversion from HOL program units to VLSI circuits. The gaps in our transformation systems are closing at Utah, and elsewhere. The SICU design system reported by Carter in Section 4 demonstrates well the powerful use of LISP as a basis for some of the important transformations needed in mapping to silicon from higher levels of abstraction. We believe significant progress has already been made, and though it may appear to be only a start, more significant

progress is expected soon.

3. VLSI Research at Utah

VLSI research at the University of Utah is being pursued actively in five areas. These are: (1) The ADA to Silicon Complier, (2) Path Programmable Logic (PPL), (3) Self Timed System Architecture, (4) Device Modelling, and (5) Switched Capacitor Filter Design. Section 5 provides a more complete overview of on-going research in areas 2, 3, 4, and 5, and describes the research interactions occurring in VLSI design at Utah. All five of the major programs complement each other and help to establish a practical and viable VLSI program of broad scope.

The Ada to Silicon Compiler effort is the principal interface with the research described elsewhere in this report. The objective of the PPL research is the development of structured logic for the design of VLSI using extensive computer aids. The self-timed system architecture program involves the development of procedures and VLSI implementations of asynchronously operating systems. (See also Section 3 which provides an introduction to the Speed-Independent Control-Unit (SICU) design system now in an advanced state of implementation.) The device modelling is a research program for measurement and verification of device parameters for computer modelling of integrated circuits. The object of the switched capacitor filter work is the development of a family of structured analog modules which use the PPL methodology for

digital control.

4. Automating the synthesis of VLSI implementations from high-level specifications.

The primary thrust of this research has been along the three directions summarized below and detailed in Section 4.

1. Development and enhancement of software tools to support our experiments in program transformation and synthesis.

   The initial set of tools that we envision include

   - a syntax-directed editor for Ada;
   - a pattern matcher that is cognizant of the syntactic primitives in the language;
   - an ability to write sequences of transformation commands and apply them to the Ada program;
   - a mechanism for maintaining the history of interactions that are carried out in the course of obtaining an implementation (both by the user and by the system), so as to be able to backup to preceding points in the developmental sequence.

   We expect the above set of tools to serve as the starting point for experimentation with the transformations of Ada programs into state machine descriptions. For interfacing with the subsequent phases of the overall transformation process, and also to provide a two dimensional picture of the evolving architecture, it will subsequently be necessary to develop an appropriate set of graphics tools.

2. Detailed case studies of the methods for transforming Ada programs into silicon, as a prelude to its automation (and with the additional objective of identifying potential problems).

   This aspect of our research has focused on hand simulation (as a prelude to automation) of the actual transformations that we envision applying to Ada programs in transforming them into State

machine descriptions. Such descriptions, may, for instance, be input to the SICU design system described by Carter in Section 3 Two examples (a bubble-sort and a roster-keeper) have been studied in detail, and this experiment will be described in a forthcoming technical report. (See also section 4.5.) The automation of these exercises is expected to be commenced as soon as the tools described above are available.

3. Research into a theoretical basis for the design of special purpose VLSI systems, with the aim of exploiting the feasibility of annotating Ada programs with formal specifications (in particular, Ada packages and tasks).

Background

This report outlines the beginning steps taken in any integrated research effort toward the development of a methodology, and supporting systems, for transforming Ada programs, or program units, (directly) into corresponding VLSI systems. The time seems right to expect good results. The need is evident; special purpose systems should be realistic alternatives where simplicity, speed, reliability, and security are dominant factors. Success in this research can lead to attractive options for embedded system applications.

Ada programs can be regarded as ensembles of machines, one per program unit (module), which in turn may be mapped directly into corresponding VLSI structures on one or more chips with interconnecting (packet switched or other) communication nets.

Many of the transformation steps, when performed manually, when optimization is not everywhere crucial, and when care is taken to constrain somewhat the structure of the source Ada program, appear to be understood.

The research reported here is part of a five-year plan, the first year of which focuses on "proving" the concepts through a realistic demonstration of the methodology for a specific example Ada program (a silicon representation of part or all of the DoD Standard Internet Protocol, IP, initially expressed in Ada.) Since the mapping from Ada to VLSI is seen as a multistep, iterative procedure, considerable effort for the following four and a half years will be

the invested in the development and tailoring of intermediate languages and their bridging algorithms (compilers), as needed, and in the development of objective criteria for their use with feedback loops for iterative design.

Implicit in these objectives is the development of a set of hardware structuring paradigms (rewrite rules) whose application can ensure that transformation steps between levels of abstraction in the design process are well structured in order to preserve the integrity and, where possible, the clarity of the original Ada specification. Some paradigms, but of course not all, lead to highly efficient implementations.

Motivation

Good system designers have claimed for years that there should be no essential difference among systems (or algorithms) that are well-implemented entirely in software, entirely in hardware, or in some mixture of the two. The choice of medium, hardware or software, it has been argued, is a matter of convenience and expediency. What has been lacking, however, is a single unifying, teachable, and practical discipline of system design which explicitly accounts for and caters to the indistinguishability of hardware and software and which, when adhered to, directly aids the designer in the implementation process. The need for such a discipline is especially urgent as the opportunity looms to transform algorithms and systems via design automation aids into silicon. Design aids must, however, be developed in the framework of an appropriate design discipline, and with the help of appropriate specification languages.

As VLSI capability has progressed, the predominant question posed by the "sideline judges" has been what systems or subsystems will be worth putting into silicon, not how to do it. This question can become truly moot if we can significantly increase our skills in transforming soft algorithms into their hardware realizations. For, once we become skillful at this art, by overlaying it with an effective discipline, a much wider range of algorithms and systems --less and less general-purpose-- can become appropriate for direct transfer to silicon.

If this objective --easier, more economical means for transferring arbitrary algorithms into silicon-- can be achieved, then task force leaders with management perspective and their system design consultants will find it more feasible to ask their programmer-logic designers to implement system components that require less, rather than more, embedded software.

Major goals of computer science and engineering disciplines in the decade ahead should be: (a) to demonstrate that design for the direct hardware realization of important systems applications or their key components is, and should be recognized as, a "programming" activity, and (b) to help make this way of system design viable and attractive as an alternative to conventional methods.

Beginning with this report, we address a major subgoal of the above, namely to demonstrate that transformation of entire Ada programs or their components into direct hardware equivalents (integrated circuits) is feasible, and to contribute in a significant way toward a methodology for automating this transformation.

A corollary, but somewhat longer-range (and dependent) goal, is the development of program-driven methodologies for testing circuits that represent high-level language modules, and for integrating them into larger subsystems. If specification of a component in Ada can be demonstrated to serve as a satisfactory starting point for the hardware design process, it can also, in

principle, serve as the starting point for generating the test sequences for the derived hardware and, if needed, for defining the interfaces to integrate derived and tested hardware into a larger, more comprehensive system.

Individual Subproject Reports

1 IMPLEMENTATION OF THE DOD INTERNET PROTOCOL IP

by

Gary Lindstrom

We summarize here the principal features of the our approach to the Internet Protocol case study. This commentary is extracted from our multi-author local working document "Internet Protocol Case Study: Background, and Initial Design" [2]. The official Internet Protocol specification may be found in [18].

1.1 Major Design Decisions for the specification of IP

Our implementation of the Internet Protocol is termed the INM (for Internet Module). There have been two major design decisions for the INM thus far:

1. First, we have structured the Ada formulation of our Internet Protocol Module (INM) into three submodules:

   a. INM_OUT, handling incoming datagram fragments from a single local net;

   b. INM_IN, handling the symmetric function for outbound datagram fragments, and

   c. INM_SRV, interfacing INM_IN and INM_OUT to the local Host(s), and providing a bridge between such submodules on connecting local nets at gateways.

2. The second decision is to use a two-phase Ada rendezvous to implement both the upper (Transmission Control Protocol Module or TCM) and lower level (local net module or LNM) interfaces. In each case, a task entry call is performed by the initiator of the data transfer action, with the receiver servicing the transfer through an appropriate accept statement. When the data transferred has been fully processed, a reciprocal rendezvous takes place (with entry call and accept roles reversed) to report the success or failure of

that processing. [An alternative formulation, based on passing messages via ports such as is done in the i432 architecture, is also under consideration.]

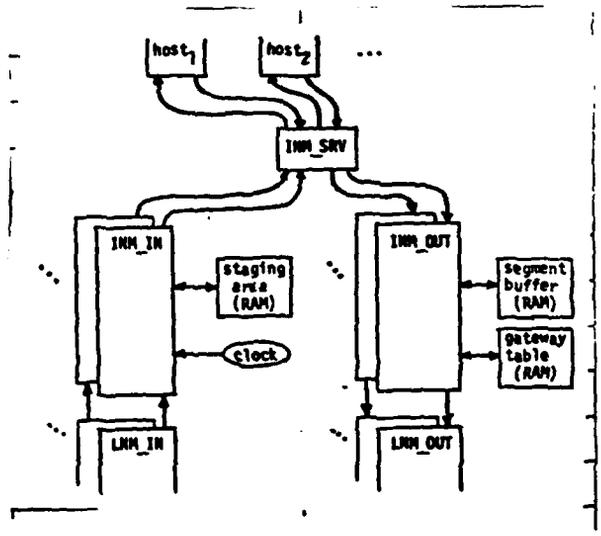

Figure 1: Internet Module decomposition.

Figure 1 (taken from [15]) sketches the structure of INM under these design decisions. These lead to the following division of functional responsibilities:

1. INM_SRV:

    a. Receive segments from and deliver segments to TCMs in the Host(s) served.

    b. Accept incoming segments from the INM_INs, and

i. deliver via TCM RECV calls all segments so addressed, and

ii. (if implementing a gateway) route to appropriate INM_OUTs all through traffic.

c. Maintain a gateway transfer table, used to route all outbound segments (whether from a local Host or neighboring INM_IN). If an outbound segment has a non-local net name in its destination address, that net name is used as a key to select the appropriate next gateway directly reachable by a local net served.

d. Implement ICMP (Internet Control Message Protocol) message generation and transfer.

e. Handle options:

i. Security: reject all [illegible]ified traffic, perhaps with an ICMP report of "[illegible] unreachable".

ii. LSRR (Loose Sour[illegible] Return Route), SSRR (Strict Source and Return [illegible]), and record route.

iii. Timestamping: (note this requires a time of day service, presumably from the TCM).

f. [Note that all message traffic through the INM_SRV is in segment form; datagram (or fragment) form is used solely within INM_IN and INM_OUT submodules.]

2. INM_OUT:

a. Form fragments from segments received from INM_SRV.

b. Deliver fragments to the LNM_OUT of its assigned local net, along with their local net addresses (destination or gateway), as provided by INM_SRV.

c. Map the Internet type of service parameter to an appropriate local net type of service, or reject fragment if this is not possible.

3. INM_IN:

a. Receive fragments from the LNM_IN of its assigned local net.

b. Reassemble fragments into complete datagrams (destination fragments only).

c. Delete overage and erroneous fragments. Note this requires a timing pulse at least once each second; this could be obtained from INM_SRV or a separate timing device.

1.2 IP Development and Testing Plan

A staged development and testing plan has been formulated for the Internet Protocol case study. This plan is based on several premises, including:

1. Testing should be done from the very earliest stages of module development, and at frequent intervals thereafter. This contrasts with the "build it all - test it all" strategy, which is in gross disrepute in the software engineering community, and which would be especially risky to us given our concurrent hardware design effort. And, after all, early simulation is one of the key advantages we assert for our overall approach.

2. The development stages should reflect realistic (and hence conceivably useful) specializations of the overall repertory of IP functions. This will permit early feedback from IP experts on the correctness of the design as it develops, since each level will be recognizable as an IP in a specialized application setting.

3. The test environment should be done exclusively in Ada, if at all possible. This will prevent premature commitment to ad hoc module interfaces, and maximize portability (especially to the i432-based hardware test bed envisioned). And, again, readability of the test environment code should permit IP experts to check its realism.

4. Finally, the test environment should present a clean and readily comprehensible picture of the system's external behavior. This standard virtue will be particularly important to us, since the testing will involve a diverse subset of our research staff. In addition, we will certainly want highly informative demonstrations to be quickly obtainable.

1.3 Plan

We now enumerate the development levels planned. For each, we itemize the underlying application model, necessary component functions to be added, and supplemental testing capabilities needed. Not listed, but implied, are implementation at each level of appropriate datagram options and ICMP messages.

1.3.1 Level I

The first level comprises the simplest conceivable function subset that could have some application utility.

- application model: a single local Host performing only SEND operations on an isolated network.
- component functions: full INM_OUT functionality, INM_SRV limited to direct segment passage from Host to INM_OUT.
- testing environment: Host and LNM_OUT interface, with simple emulation controlled via terminal I/O.

1.3.2 Level II

- application model: a single local Host performing SEND/RECV operations on an isolated network.
- component functions: full INM_IN functionality, INM_SRV expanded to deliver incoming datagrams to Host.
- testing environment: LNM_IN interface, emulated from terminal, for composing and "delivering" incoming fragments from local net; Host interface extended to RECV as well as SEND segments.

1.3.3 Level III

- application model: a single local Host performing SEND/RECV operations on a network with gateways.

- component functions: INM_SRV extended to contain a gateway table (with downloading from Host as pseudo-segment).

- testing environment: Host interface extended to accept gateway table initialization and change "transactions".

1.3.4 Level IV

- application model: an IP at a gateway, supporting SEND/RECVs from a single local Host as well as gateway traffic.

- component functions: INM_SRV extended to manage multiple INM_IN and INM_OUT pairs, and provide an exchange connection for in-transit datagrams.

- testing environment: LNM_IN and LNM_OUT emulation code extended to deal with multiple instances and interleaved interactions among them.

1.3.5 Level V

- application model: an IP at a gateway, supporting SEND/RECVs from multiple local Hosts, as well as gateway traffic.

- component functions: INM_SRV extended to interface with multiple Hosts.

- testing environment: Host emulation code extended to deal with multiple instances and interleaved interactions among them.

1.4 Schedule

Our Year One goal is to complete software development and testing through Level III.

We feel a reasonable Year Two fabrication goal will be to produce a silicon version of the full INM_OUT submodule. This will require a stable Ada version of INM_OUT by the close of Year One, i.e. one acceptable from both IP correctness and hardware realizability considerations. Our confidence in thus freezing the INM_OUT design will be enhanced by having completed software development through Level III at the time this binding is done.

2 IDENTIFYING ADA PROGRAM STRUCTURE CANDIDATES FOR TRANSFORMATION INTO SILICON

by

Elliott I. Organick

Here we distinguish between two questions and focus primarily on the second of these. To simplify the phrasing and discussion of these questions we use the term "customization" to refer to the transformation to silicon of an algorithm expressed in Ada.

The first question is: Why customize an algorithm? Given an algorithm that is already chosen for customization, we then ask the second question, which is: How should the algorithm be structured as an Ada program to facilitate its customization, and how can it most easily be tested and integrated into a larger system? An important corollary to both these questions is: Given an existing Ada program structure, can we identify those parts of it that can be relatively easily customized?

An algorithm, which may be expressed as part of a larger Ada program, becomes a candidate for customization when important speed, integrity, or security considerations prevail.

- Customization can lead to speed improvements in two ways:

    * by achieving concurrency increases at any of several semantic levels, from the level Ada multitasking down to the levels in the hardware implementations of single states in the machines that implement individual units of the customized part of the Ada program.

* by eliminating "system overhead steps". These are steps that are normally compiled into an algorithm or are provided either in the form of run-time software, firmware, or hardware of the supporting computer system. Such overhead actions, usually considered essential, are executed to ensure the correctness of the system as a whole, rather than of the specific algorithm that needs to be customized -- for example, protection and memory management functions in conventional computer systems.

- Customization can lead to fixing the algorithm so it can be counted on as "constant". Such constancy leads to greater integrity of the system as a whole.

- Customization can lead to greater security of the private parts of a customized algorithm, since it is physically isolated from the rest of the system and hence can be more easily protected from accidental or deliberate damage; it is more difficult to copy, alter, or mimic a customized algorithm.

We elaborate no further on the above arguments for customization and instead proceed to the second matter of identifying Ada program structures that are useful in the customization process. Our second question was how an algorithm should be structured as an Ada program to facilitate its customization, its testing, and its integration into a larger system. To answer this question we first review certain special attributes of a high order language like Ada. Five ideas are germane in this context. For this discussion, a "program part" is an Ada syntactic unit which contains an interface specification (e.g., a package, subprogram, or task):

- Ada offers a framework for specifying structural relationships among program parts.

- Ada offers a framework for specifying communication paths among program parts.

- Ada permits us to specify what should be private or hidden within

individual program parts.

- Ada permits us to describe the logical behavior of individual program parts.

- Since an Ada program part has a fixed interface, not only is its body-part implementation independent, but the entire part is potentially device independent; that is, it can be implemented in different media (e.g., in custom VLSI circuitry) without change in essential semantics.

2.1 Ada program graphs viewed as ensembles of "engines"

An Ada program may be thought of as representing a system. One can represent the program, and the system it "models", as a graph. The nodes of the graph map to specific abstract or real machines called engines, and the arcs of the graph map to communication paths between these abstract or real engines. Communication between program parts, as implied by each arc, follows a particular discipline. If we think of the nodes of a program graph as the program's "parts", then any one of the parts can be "state-bearing" in a significant way; that is, any part, such as a package can "own" data and the counterpart engine can have sole control over a local store that holds the owned data.

Communication disciplines that can be expressed in Ada spans a wide discipline, wider that perhaps one ordinarily infers, say from a casual reading of the Ada Reference Manual [1]. These disciplines range from fully synchronous subprogram calls on package operators, to partially synchronous task entry calls using the "rendezvous" mechanism, to fully asynchronous inter-task

messages in cases where Intel 432-styled "communication port" operations can be used.

The Ada program style demands and imposes a sharp division between specification and implementation of individual program parts. A part specification fixes its interface with other parts and that interface is necessarily independent of the medium used for the part's implementation.

Ada encourages the composition of solutions whose structures suggest distributed computing realizations. The graphs of Ada programs can be partitioned into sub-ensembles (sub-graphs) divided by what we call "minimal sharing cuts", that is, cuts that denote a minimum amount of information sharing between elements of separated sub-ensembles. Designing an Ada program whose graph exhibits sharp minimal sharing cuts is pragmatically motivated and not logically essential. When such cuts can be built into new program graphs, or discovered in existing program graphs, then one or more sub-ensembles become candidates for customization. They become what we shall call "Special Function Architectures" or SFAs, a term used recently by Rattner in a related context [19]

This concept is illustrated in Figure2. Any minimum sharing cut that divides a portion of the graph representing parts that will be executed on general purpose host computers from the parts to be customized may also be viewed as a hardware bus.

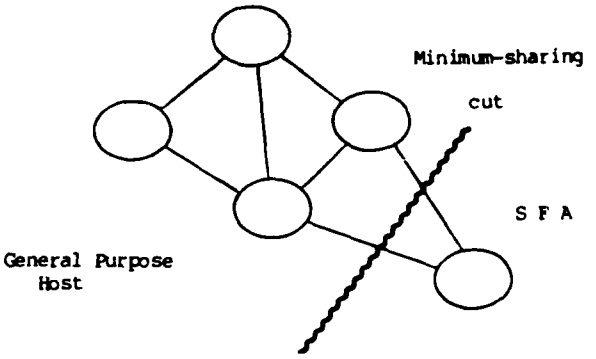

Figure 2: Ada program graph divided into two subensembles by minimal sharing cut. One subensemble executes on a conventional General Purpose Host; the other is directly realized in circuitry as a Special Function Architecture. The cut can also be regarded as a hardware bus.

In our design methodology, each customized part, which we now call an SFA, must retain the semantics of the Ada program unit from which it is derived. In particular, it must have the same interface as its original software abstraction. Adherence to this principle automatically gives us a discipline for testing a customized part P at a high level, since P must per force behave identically, except for timing, with the conventionally compiled program unit

that P replaces in the larger program.

The SFA functions internally as, what Rattner has called, a "hardware process". That is, an SFA can be represented as a separate machine or engine or by a set of such engines --if need be. More specifically, by an "engine" we mean a pair of components consisting of a state machine and the "data path" it controls and senses. Since the SFA is technology independent, it should be possible, in principle, to implement it in any circuit family.

The implementation strategy for an SFA is somewhat open. More than likely, its interface with related units is asynchronous. Also, its "internals" can be represented by one or more asynchronously communicating engines. However, each engine may itself be implemented with either a synchronous or an asynchronous control unit. (See the next section of this report for a description of our work on design of speed-independent control units.)

2.2 Mapping Ada program units to SFAs

For our current research we limit what we mean by "program units" to

Ada Library-level Packages that contain no embedded Tasks

or to

Ada Tasks that have no globally accessible objects.
(Syntactically, each such task would be declared within a library-level package which contains no other declaration but that of the task in question.)

It is conceptually easy to transform such Ada program units into ensembles of

synchronous or asynchronous engines using meaning-preserving rewrite rules. Given no silicon area constraints and modern interactive software tools and graphics facilities, the job of customizing a program unit is doable.

Even though "doable", and even though Ada seems well suited for specifying SFAs that reduce to relatively non-regular data and control structures, several related (hard) questions remain. Two of these are:

- How practical is it to perform customization of an Ada program unit to yield a high degree of concurrency in the VLSI implementation? (Investigating such optimization, while considered to be within the scope of our research, is not pursued here.)

- A related question is: Are Ada programs over-specified for purposes of reducing them to silicon? We recognize that starting with an algorithm expressed in Ada, rather than an abstract (e.g., axiomatic) specification may in some cases have significant drawbacks. On the other hand, using an Ada specification appear to have advantages relative to concreteness and testability.

2.3 Top-level system structuring rules for SFAs

Central to our transformation methodology is the observation (or guiding principle) that there exists just a small number of what we call "top-level, Ada-specified, structuring rules" that are applicable for driving the transformation to an SFA. The two most important of these rules are:

1. Program unit interfaces imply specific, but high-level communication protocols. Such protocols are governed by Ada semantics and more precisely by the particular "role"s of the individual program units.

2. The interior of an SFA is derived from the declarations and statements of the program unit as follows:

   Its data path is derived mainly from the declarations and may be

regarded as a collection of instantiations of generic data abstractions, which include storage and related functional units.

Its control or state machine component is derived mainly from the control flow implied by executable statements of the program unit.

(Whether the ultimate representation of the control component is centralized, as in conventional state machine designs, or distributed throughout the data path, as in some data flow schemes [13], is regarded as a secondary issue. Whether the control part is speed-independent or synchronous in style is regarded as an implementation issue.)

We elaborate here on the first of these two structuring rules by illustrating the kinds of roles played by Ada program units. First we examine the roles of Ada packages and then the roles of Ada tasks.

2.4 Roles of Ada packages

Each package maps at the top level to a single engine, i.e., to a single (state machine, data path) pair. Except for activity during its initialization, the engine "idles" until receipt of a call on one of its public operations. A package can perform one of two roles. We call these the "owner" and "transformer" roles:

- An owner package, which is history sensitive, manipulates its own objects; these are hidden from other program units. Owned data may be correctly accessed under contention only if operations of the package are reached via an arbiter unit. We assume that the customized version of an owner package has an associated private store to hold instances of owned objects. Hence, operation requests to owner packages would generally involve the transmission of relatively short packet sequences (containing information to identify and to update the owned objects but not the objects themselves.)

- A transformer package, which is history insensitive, owns no data

objects. It merely manages objects; e.g., manipulates objects specified by the caller or creates objects on behalf of the caller. A transformer package can be replicated, in case it must be accessed by more than one engine concurrently. The customized version of a transformer package requires only enough private store to hold local variables, such as for an instance of the object it currently manipulates. On the other hand, operation requests to transformer packages would generally involve the transmission of relatively long packet sequences (because such sequences must contain not only the information for updating an object, but also object itself.) For this reason, one would expect that customization of transformer packages would be generally less attractive than customization of owner packages. However, one can imagine customizing a transformer package when the objects that the package transforms are small or when the package to be customized will be embedded in the customization of an ensemble of other program units. In the latter case, transmission of relatively long packet sequences per operation call could be tolerable if the transmission is confined to local bus structures within the SFA representing this ensemble.

Regardless of the role, owner or transformer, any package can be "promoted" to a task, and thereby serve as its own arbiter.

2.5 Roles of Ada tasks

Ada tasks can contribute to the concurrency of the overall system in which they are embedded. Two forms of concurrency are achievable:

- Independent Server Tasks can contribute a parallelism effect. A Server Task can execute useful work in parallel with its requester by performing request-related work immediately after releasing the requester from a rendezvous. Concurrency is maximized by making the "acknowledge" (i.e., completion of the accept statement) as early as possible.

- Server-Requester Tasks can contribute a pipelining effect. A Server/Requester Task accepts requests and also issues requests (task entry calls) to other tasks. Such a task, B, can serve as an element of a pipeline by accepting a task entry call from task A and, after acknowledging (accepting) A's request, transmitting a related request

for service to task C. Upon receiving C's acknowledge, B then returns to the state where it is ready to accept another request from A. (We are especially sensitive to the potential for pipelining in our current research for realizing SFA versions of the DoD Internet Protocol whose structure is described in the preceding section.)

In addition to contributing to the concurrency of the overall system, an Ada task can play two types of explicit synchronizing roles:

- As a multi-task synchronizer. When a set of tasks S must be synchronized so that each member of S is allowed to proceed only after being notified of some event of common interest, another (server) task T may act as a synchronizer for the members of S. (In Ada, this is done, for example, by nesting in T accepts of entry calls from each of the members of S.

- As an arbiter for operation calls on one or more packages. For example, an "owner" package P cannot backlog the operation calls made on it, but a Server Task can keep a backlog of entry calls made on it. Hence, any Server Task T may act as an arbiter of requests on P by accepting them one at a time as entry calls and then forwarding them in lock step as operation calls on P. It can be seen that a Server Task can also be structured to arbitrate calls on more than one package.

Because they play so many useful roles, Ada Server and Server/Requester Tasks will be regarded as good candidates for customization in many applications. Here we assume that Ada programmers who model systems will make full use of the Ada task in decomposing algorithms that model real world entities. If so, there will be many Ada programs whose structures will, with little or no modification, reveal program unit ensembles that are candidates for customization.

We offer, in closing this section, two observations regarding the circuit realization of Ada tasks. A Server Task can always be represented by at most a

(base) pair of engines; we call these the "Front" and "Back" engines, respectively. (To gain added concurrency within the Task's circuit realization, it may be desirable in practice to perform additional decomposition into more than two engines.)

- The Front engine serves as the manager of the task's entry queue(s). This engine implements directly the synchronization function of the Ada rendezvous, acting as a "broker" between requester program units and the main server logic of the task. (The Front engine is logically connected to the "system bus" interface of the SFA.) The Front engine logic obeys the semantics of an Ada task interface. Because these semantics are standardized, some of our research involves classifying this semantics (that of the Ada select statement in particular) into a tractable number of prototypical categories or paradigms. Such a classification is expected to help us simplify mapping the interface for an individual Ada task into the circuitry of a Front engine using a macro-expansion style of mapping.

- The Back engine in all cases represents the specific logic for the particular Server Task being customized.

3 THE SPEED-INDEPENDENT CONTROL-UNIT DESIGN SYSTEM

by

Tony M. Carter

A Speed-Independent Control-Unit Design System (SICU), is under development as a software tool for aiding systems designers in the simulation and layout of control-units. This software tool may well play a key role in the overall Ada to silicon transformation system now under development at Utah. For the design system reported in this section, control units are implemented using the One-Hot technique proposed by Hollaar [10]. The simulation is done interactively and compilation to integrated circuit composite is automatic.

The SICU system is perhaps the first attempt to integrate the design of speed-independent control-units into a CAD system. The most important features of this research are the use of Path Programmable Logic (or PPL), a derivative of the SLA [16, 22, 23, 24, 25, 26], as an integrated circuit implementation technique, the embedding of the software in a PSL (Lisp) [4, 5, 6, 7] programming environment, and the automation of compilation to integrated circuit composite layouts.

PPL has been described as an integrated circuit implementation technique which readily allows and encourages the use of CAD in the design of integrated circuits [26]. Until the birth of SICU, CAD for VLSI design using PPL was virtually non-existent. A Master's Thesis in preparation at the University of Utah by Tony M. Carter presents some of the history of CAD tools for PPL (SLA)

design and elaborates to a great extent on the design, algorithms and implementation of an SICU.

It has already been shown that PPL directly reduces the time needed to layout a medium size system from man-months to man-days [26]. SICU advances the art by another significant step by reducing layout time for control-units from man-days to computer-minutes.

The SICU consists of two major modules: a Functional Simulator and a Silicon Compiler.

3.1 SICU Features

SICU operates on a language known as SICUDL (Speed-Independent Control-Unit Description Language), designed at the University of Utah by Tony M. Carter and specifically tailored to the representation of speed-independent control-units implemented using the One-Hot technique. It includes such features as:

1. Multiple state-machine control-units,
2. Global input declarations,
3. Sequential control flow,
4. Conditional branching of arbitrary degree from any state (multiple transitions),
5. Initiation of concurrently executing control paths (FORK),
6. Termination of concurrently executing control paths (JOIN),
7. Unconditional outputs during residence in states or during transitions (i.e. both Moore and Mealy machines),

8. Conditional outputs during residence in states or during transitions, and

9. Enduring (latched) or ephemeral (gated) output generation.

Figure 3 illustrates the linguistic representation of a simple, three state control-unit.

```
control unit Example:

  statemachine Example1:

    startstate S1:       % S1 = State 1
       moveon I1 to S2; % I1 = Input 1, S2 = State 2
       hold O1;          % O1 = Output 1 (gated)
    end;

    state S2:            % S2 = State 2
      moveon I2 to S3;  % I2 = Input 2, S3 = State 3
      set O2;           % O2 = Output 2 (latched)
    end;

    state S3:            % S3 = State 3
      moveon I3 to S1;  % I3 = Input 3, S1 = State 1
      reset O2;         % O2 = Output 2 (latched)
    end;
  end;  % State Machine Example1
end.    % Control Unit  Example
```

Figure 3: SICUDL Program for a Simple Control-Unit

3.2 Simulator

State-machines implemented using the One-Hot technique are essentially token passers (as in Petri Nets); hence, simulation can be viewed as observing token flow through the network described by equating states of a state graph with "places" of a Petri net and arcs of a state graph with "transitions" of a Petri

net.

The Functional Simulation of Speed-Independent Control-Units helps the user to verify the structure of the control-unit in relation to its input/output behavior. Such simulation is functional in nature. When speed-independence is forced upon the implementation, the functional simulation also becomes valid also as a logical circuit simulation which generates sequence information (although no timing information is generated.) Figure 4 illustrates the type of display used during simulation. The first column shows the state values, the second the input values and the third the output values.

```
Control Unit EXAMPLE

State Machine EXAMPLE1

S1      = 0     I1      = 1     O1      = 0
S2      = 1     I2      = 0     O2      = 1
S3      = 0     I3      = 1
```

Figure 4: Sample Screen Image During Functional Simulation

3.3 The Compiler

The second major module of the SICU system is the compiler. It will convert a high-level description of the control-unit, such as illustrated in Figure 5, into an integrated circuit composite layout using Path Programmable Logic [25] as the integrated circuit implementation technique. PPL appears to be a good circuit implementation choice for speed-independent control-units; routing software is much simpler than for more conventional layout schemes.

```
 : - - - - - : - -
| |-|-|-|-|-| |-|-|
|0|- - - - -|0|- -
|1|-|-|-|-|-|2|-|-|
| |- - - - - L|- -
| |-|-|-|-|-| |-|-|
 : - - - - - : - -
|-|-|-|-|-|P-S-1|-|
 : - - - - - : : -
|+-1-P|-|P-1-R|-|-|
 - : - - - : - : -
|   | | |   |   | |
|   |I|I|   |   |I|
| S1|1|2| S2| S3|3|
|   | | |   |   | |
|   | | |   |   | |
 : : : : : : : : :
---S-----R---R-----
 : : : : : : : : :
|-|1-1-P---S-0|-|-|
 : : : : : : : : :
|R---P-----1|-|-|-|
 : : : : : : : : :
|0---P-1---1---S|-|
 - : : : : - : : :
|-|-|-|-|R-P---1|-|
 - : : : : - : : :
|-|S---P-0-----1-1|
 - : : : - - : - :
|-|1---P-----R|-|-|
 - - : : - - - - :
```

Figure 5: PPL Representation of Compiled Control-Unit in Figure 3

The SICU design system currently performs only row-wise compaction of the generated PPL program by splicing row segments together. A study is currently being made of the benefits of input replication and column-wise compaction of PPL programs on the design and implementation of speed-independent control-units and their impact on software systems that implement both the

compiler and the functional simulator.

3.4 Circuit structures

The SICU compiler essentially maps the state-transition diagram onto silicon (via PPL) as follows:

- One data latch (8 transistors) per state,
- One PPL inverter (3 transistors) per input,
- Two PPL row segments per transition,
- One PPL row segment for all unconditional outputs controlled by a state or transition,
- One PPL row segment for each set of outputs contingent on a given condition controlled by a state or transition,
- One PPL inverter (3 transistors) per ephemeral output, and
- One PPL latch (4 transistors) per enduring output.

3.5 Contrast with related work

With our SICU design system we attack the VLSI problem from a different perspective than most silicon compilers, regarding our approach as anticipating both present and future needs. Several other "Silicon Compiler" projects, mostly by DARPA-sponsored research groups, are under way. Most of these projects operate on the assumption that control-units are easy to implement using the PLA. This assumption has driven most of these projects to concentrate on data-path generators for fully synchronous systems using more or less custom circuit design techniques. Such projects include the LAVA project

at Stanford [14], the Data Path Generator used for Scheme-81 at MIT [21], and the SSP project at Caltech. A major problem with these data-paths is their almost universal clocked nature. This perhaps unconscious decision to use entirely clocked logic appears to be based on the assumption that on-chip clocked systems will remain feasible in the long run. While it would appear that these other silicon compiler projects will produce useful algorithms and software systems, it is felt that an effort will have to be made in the generation of speed-independent data-paths.

Since control-units implemented using the PLA are usually synchronous (and since it remains to be seen whether either traditional or contemporary asynchronous circuit design techniques can be applied to the PLA), it is felt that the synchronous PLA should be either developed to the level of sophistication found in PPL or be regarded as ineffective as an implementation technique for speed-independent control-units. The decision to use synchronous systems has mainly been driven by the large device sizes and high power consumption requirements of NMOS integrated circuits. Emerging CMOS technology promises adequately low power consumption for asynchronous circuit designs to be competitive. C. L. Seitz of Caltech has postulated that as integrated circuit device dimensions scale down, timing will become such a problem that working clocked systems will be impossible to design [20]. For this reason, research directed toward development of techniques for implementing speed-independent systems and control-units seems appropriate.

3.6 Future work

Much remains to be done in the definition and implementation of our SICU design system. Results so far are encouraging and indicate that the design of speed-independent systems should be both feasible and perhaps easier than the design of synchronous systems.

4 AUTOMATING THE SYNTHESIS OF VLSI IMPLEMENTATIONS FROM HIGH LEVEL SPECIFICATIONS

by

P.A.Subrahmanyam

In this section, we delineate our research aimed at automating methodologies for synthesizing special purpose VLSI circuits from high level specifications. This work falls into the following categories:

1. Development and enhancement of software tools to support our experiments in program transformation and synthesis.
2. Detailed case studies of the methods for transforming Ada programs into silicon, as a prelude to its automation (and with the additional objective of identifying potential problems).
3. Research into a theoretical basis for the design of special purpose VLSI systems, with the aim of exploiting the feasibility of annotating Ada programs with formal specifications (in particular, Ada packages and tasks).

We now elaborate on some facets of this research in slightly greater detail.

4.1 A Software Environment for Experimenting with Ada-to-Silicon Transformations

To support our experiments in transforming high level specifications of problems into efficient hardware structures, it is necessary to have an appropriate complement of software tools. The initial set of tools that we envision include

- a syntax-directed editor for Ada;

- a pattern matcher that is cognizant of the syntactic primitives in the language;
- an ability to write sequences of transformation commands and apply them to the Ada program;
- a mechanism to the history of interactions that are carried out in course of obtaining an implementation (both by the user and by the system),so as to be able to backup to preceding points in the developmental sequence.

We expect the above set of tools to serve as the starting point for experimentation with the transformations of Ada programs into state machine descriptions. For interfacing with the subsequent phases of the overall transformation process, and also to provide a two dimensional picture of the evolving architecture, it will subsequently be necessary to develop an appropriate set of graphics tools.

The primary implementation environment we are using is that provided by Interlisp [11]. Integrated into the Interlisp is a system called POPART, which is a Preducer Of Parsers And Related Tools. The purpose of POPART is to produce parsers tailored to Interlisp conventions for grammars specified in a BNF variant which permits regular expressions. Given an Ada program, for example, the parser produces parse trees in Interlisp in a relatively data representation independent way. Together with the parse tree, it also produces a pattern matcher and instantiator, editor, pretty printer, history manipulations of interactive commands and a rudimentary program transformation mechanism. Using POPART, we have produced, and are debugging, a set of tools

tailored to Ada.

After the debugging of this initial set of tools is complete, we will focus on further enhancement of the transformational implementation environment. Some effort will be concentrated on providing graphics tools. We envision some aspects of the parser having to be rewritten in order to rectify some of the unsuitable characteristics of the extant system.

4.2 Transforming Ada Programs into State Machine Descriptions: Two Case Studies

This aspect of our research has focused on hand simulation (as a prelude to automation) of the actual transformations that we envision applying to Ada programs in transforming them into State machine descriptions. Such descriptions, may, for instance, be input to the program written by Tony Carter (cf. discussion in a previous section of this report). The author is being aided by Sanjay Rajopadhye in this effort. Two examples have been worked out in detail, and this experiment will be described in a forthcoming technical report. The automation of these exercises is expected to be commenced as soon as the tools described above are available.

We first recall that the overall transformation strategy underlying the current effort can essentially be partitioned into three major phases:

1. Programs in a stylized subset of Ada are transformed into a state machine description in an intermediate language.
2. The state machine descriptions are mapped into Storage Logic Arrays (SLAs) [17] (or PPLs).

3. The Storage Logic Array descriptions are converted into CIF/CALMA files describing a set of masks for the final circuit.

The declarative part of an Ada program may be viewed as defining an "environment" in which the program statement sequences define a (set of) interacting state machines. The initial phase of the transformation consists of transforming programs in an Ada subset to <state machine, environment> pairs described in a hardware description language (which is also an Ada subset). The primary tool we are using in this phase is a transformation system (POPART [28]) based in INTERLISP.

This section outlines the two example Ada programs (a bubble sort, and a roster-keeper) that have been transformed into a state machine description in a hardware description language (specifically, MODAL [8]). The immediate objective of this exercise was to get an overall picture of the transformation process by carrying out the transformations manually, but in a systematic manner.

4.3 Transformational Implementation Strategy For Phase I

4.3.1 Choosing Storage Elements

The information used in carrying out the transformations falls into one the following categories:

1. Obtainable directly from the parse tree without any global analysis.
2. Obtainable from the parse tree, but requiring global analysis.

3. Obtainable from requirements imposed externally, or from the other phases of the transformation process.

4. Obtained interactively from the user.

The initial experimentation will serve to indicate which categories some of the major Ada constructs fall into.

The declarative part of an Ada module serves, for the most part, to guide the selection of storage elements in the hardware design. A global analysis of the input program yields information that is used in deciding from amongst the design options that are available. The information so gathered includes facts such as the number of (simultaneous) accesses into an array and the number of different variables used to access an array (to determine the addressing structure of an array of registers or RAM); the number and 'nature' of comparisons (to decide between the use of multiplexors versus comparators), etc.

4.3.2 Circuit Generation

The "program" part of an Ada module serves to guide the construction of the state machine that will then be transformed into either a synchronous or an asynchronous hardware module. After the representation of the data structures has been decided upon as indicated above, a grammar driven transformation process is used to generate an intermediate circuit representation.

Conceptually, this circuit may be viewed as a graph, the nodes of which

correspond to states and detail the actions to be performed in that state; the labels on the arcs are the predicates which cause the associated state transitions. The graph so produced is then scanned to determine the exact list of control signals that are needed for each node -- these signals control the actual data transfers, or determine specific operations to be performed. Variables serving purely as loop control flags may be eliminated by using control signals instead.

The intermediate representation for the state machines can either be a stylized subset of Ada (such as suggested by Drenan in his M.S. thesis proposal), or an other suitable language. Our current examples use an existing hardware description language (MODAL [9]).

4.4 Example I: A Simple Sort Machine

The first example we discuss is the transformation of a simple Ada program that embodies the bubble sort algorithm into a state-machine that executes this algorithm. There are two distinct approaches to the problem and it was felt that working through both of them would yield valuable insight. In the first instance, we decided to perform a naive transformation i.e., 'blindly', without caring for any optimizations. In the second attempt, an attempt is made at optimization and to develop methods to gather the required information automatically.

```
procedure BUB_SRT (A:in out array(0..1023)of integer) is
        n               :constant integer := A'LAST;
        j,k,temp        :integer;
        sw              :boolean;
begin
        for j in 1..n-1 loop
            if A(j) > A(j+1) then
                temp := A(j+1);
                A(j+1) := A(j);
                k:=j;
                sw:=false;
                while (k>1)and(sw=false) loop
                    if A(k+1)>temp then
                        A(k) := A(k-1);
                        k := k-1;
                    else sw := true
                    end if;
                end loop;                          __ while
                A(k) := temp;
            end if;
        end loop;                                  __ for
end BUB_SRT
```

4.5 Unoptimised Synthesis

We first discuss the results of attempting to transform the above Ada command statement by statement into a state machine; in essence, the circuit is built as things are discovered in a sequential pass over the program.

4.5.1 Choosing Data Representations

'A:array(0..1023)of integer' is represented as an array of registers. Since no global analysis is performed we have to use single addressing into the array, as a default, with multiplexors (mux's) to resolve future multiple accesses, if any.

' n:constant integer := A'LAST ' indicates that n is an integer constant , initialized to the size of A. This is represented as a simple register.

sw would be a single bit flag.

j,k,temp are represented as 16 bit registers by default (actually the default size would, in general be a parameter of the transformation system). In the absence of any global analysis the system would not know that j and k are used only for addressing A (and hence deduce that only 10 bits would suffice). Another default assumption that is made is that all registers to have inc dec clr and ld signals.

for j in 1..N ==> initially load j thru const 1 (or clr and inc) and later compare with N.

if A(j) > A(j+1) ==> sixteen bit comparator with one input from A and one from a temporary register -- say, T1.

Choice: Which one of A(j) and A(j+1) should be stored in T1. Since j is the controlling variable of the loop it seems(?) to be advisable to keep it as free from corruption as possible. Since it is not known how j is used in the later parts of the loop (no global analysis) this choice would be arbitrary -- say, A(j) is stored first (into T1) so there should be a bus connecting A and T1. A 16 bit comparator with one input from T1 and one input from A is to be used. Since the output of A is to go to more than one place a mux is definitely

required.

The local environment may now be obtained from these considerations. We omit the details of the rest of the transformation steps.

4.5.2 Modal description

MODAL [9] is a discrete event description and simulation language, and is suitable for digital hardware description. It is based on a set of mathematically well defined functions which can be used in a general manner so that arbitrary logic bases can be described. Although this exercise was initially carried out using MODAL as the target specification language, an Ada subset, or any other appropriate SymbolTable of linguistic constructs may be used in its place.

A Modal program is modular in structure, and describes a number of units (called modules) and their interconnections. Modules may be hierarchically structured in much the same manner as in any block structured language. There are explicit scoping rules which are similar to Algol i.e. visibility is 'from the inside to the outside'. A module is first defined in one place, and if necessary, it is instantiated later. This is analogous to defining a program, function or procedure, and then calling it. While instantiating it, the actual input and output 'ports' get specified and this, in turn, specifies the interconnection of modules. A module (eventually) consists of the basic functions in the logic base, their "application"s , and certain "functional

forms" that can be derived from the function specification.

The Modal description of the sort machine is as follows.

```
module BUB_SRT [A[0..1023,0..3](p.in N[0..9],start,clk
                                p.out finis)
%It seems that clk and finis are 'universal' signals -- present in
every m/c %
module mux[0..9] <= (select selsig
                +0+     k[0..9]
                =1=     j[0..9]
%It is expected that the system would have the details and knowledge
about multiplexers stored in its database.%

module
        comparator[gt] (p.in inp[0..9] inp2[0..9])

%       Body of comparator module would be taken from the
system's knowledge base and plugged into the module.
```

Since more than one instances of mux's are seen to be required (from global analysis) it would actually have been advantageous to have a formal affix list for the mux module and instantiate it as and when required.

Assignment statements would correspond to busses in the modal program because they specify direct interconnection of circuit elements. This would complete the specification of the modules in the environment part of the circuit.%

```
%COMPONENT INSTANTIATION%

register AUX[0..3]      %to contain A(k) and A(j)%
           (p.in load)
         TEMP[0..3]     %for A(k-1) and A(j-1)%
           (p.in load)
         k[0..9]        (p.in ld, inc, dec)
         j[0..9]        (p.in ld, inc)
         N[0..9]

comparator
        comp1 (j[0..9],N[0..9])
        comp2 (mux2,TEMP[0..3])
        comp3 (k[0..9],'1')

multiplexor
        mux1 (k[0..9],j[0..9],sel1)
        mux2 (A[mux1[0..9]],AUX[0..3],sel2)
        mux3 (TEMP[0..3],AUX[0..3],sel3)
```

4.6 A Theoretical Basis For The Synthesis of Special Purpose VLSI Systems

The need for design paradigms wherein the choice of a final implementation medium --hardware or software-- is not fixed a priori, mandates that a uniform perspective be adopted both in the specification of the problem to be solved and in the development of an implementation. Ideally, this suggests that a formal, representation independent specification close to the user's conceptualization of the problem be used as a starting point. In this context, we have been investigating the ramifications of a new paradigm for synthesizing special purpose circuits directly from such formal specifications (proposed by the author [27]), on the transformation of Ada programs to Silicon.

While there are promising axiomatic specification techniques being investigated, the average (current day) system designer is not yet adept at writing formal specifications. Moreover, it is sometimes not easy to specify a desired behavior axiomatically because of inadequate knowledge about the problem domain. In such cases, it may not always be feasible to invest the extra effort required to acquire such knowledge due to project deadlines to be met. Consequently, an alternative specification technique that is viable in such circumstances may be attractive, and can generally take the form of an understandable "specification" in a high level language.

In order to enable to a smooth transition between program fragments in Ada

and formal specifications, we are exploring the use of ANNA. ANNA, an extension to Ada [12], is designed to provide formal annotations to Ada programs. This affords a choice between (1) proceeding from a representation independent specification for a problem, and then developing an Ada and/or VLSI implementation that is consistent with these specifications, and (2) "specifying" a problem by writing a more or less representation dependent Ada program for it (and optionally formally annotating the program later).

The existence of formal specifications for a problem can greatly increase the reliability of software and/or hardware developed for it by (i) precisely defining the interfaces of the programs developed; (ii) aiding both manual and/or automated verification; and (iii) enabling the development of automated tools for synthesizing implementations of the specifications. We are proposing extensions to ANNA that aid (1) the formal specification of tasking features in Ada by using temporal logic based constructs, as well as (2) the specification of performance criteria desired of modules. In addition, we are exploring a method for automatically obtaining the implementations of Ada packages bodies directly from the formal specifications of the visible annotations of a package. Techniques for automating the synthesis of Task bodies are also being investigated.

5 THE VLSI RESEARCH PROGRAM AT UTAH

by

Kent F. Smith

The VLSI group at the University of Utah is actively working in the five areas shown in Figure 6: (1) The ADA to Silicon Compiler, (2) Path Programmable Logic (PPL), (3) Self Timed System Architecture, (4) Device Modelling, and (5) Switched Capacitor Filter Design. This Section provides an overview of ongoing research in areas 2, 3, 4, and 5. These five research efforts complement each other and help to establish a practical and viable VLSI program of broad scope.

Path Programmable Logic (PPL) is an acronym used to describe a structured logic methodology. Papers on this subject can be found in [23, 22, 24, 3]. The PPL is really a derivative of the Stored Logic Array (SLA) which was invented by Suhas Patil [16]. The SLA, in turn, is a derivative of the Programmed Logic Array (PLA). The SLA is different from the PLA because of the folding of the AND and OR planes, the placement of arbitrary memory cells anywhere within the array, and the notation. The PPL is different from the SLA because of the level of physical implementation of the SLA cells. PPL cells are designed at a very primitive level and allow the definition of many different SLA cells as macros rather than being constrained to a fixed set.

The PPL research has a direct bearing on the Ada to silicon project because the fundamental building blocks which are being used for the construction of the special function architectures are the PPL cells and circuit

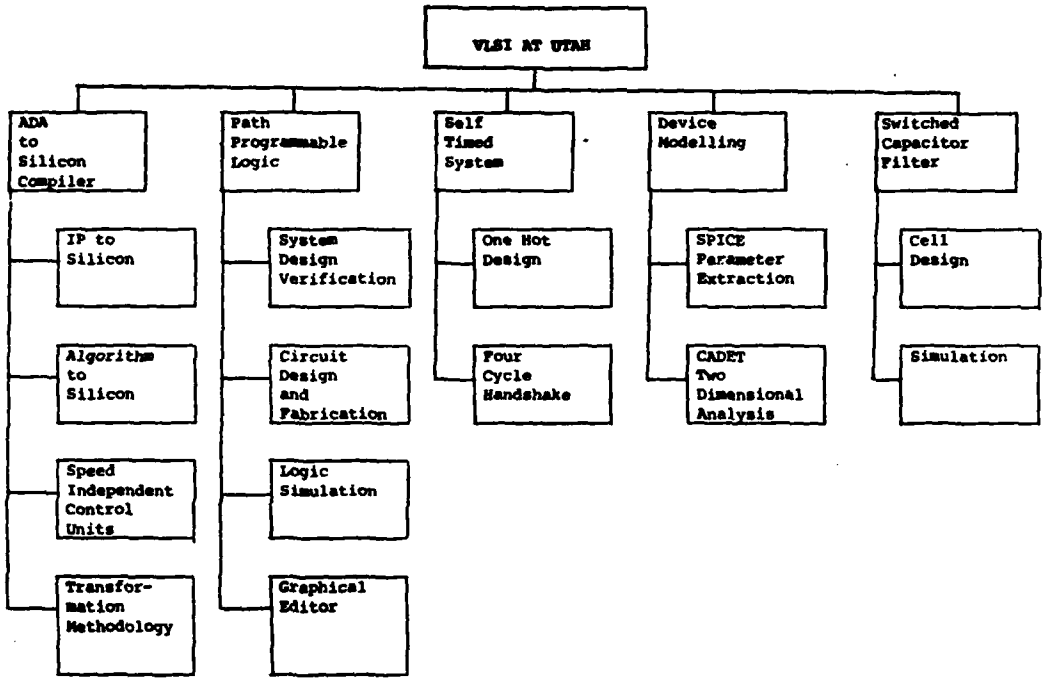

Figure 6: VLSI Research Program at the University of Utah

implementation. These techniques represent essentially the "assembly" language component of the "silicon compiler" and will be used for a "bottom up" design of the special function architectures.

The goals of the PPL program are the development of a design methodology which will result in VLSI integrated circuits having a packing density which approaches custom designs but which require design times that are one to two orders of magnitude less than the present custom layout techniques. An overview of the status of the work is described in a paper which will be published in April, 1982 in a special issue on VLSI of the IEEE Journal of Solid State Circuits. The paper is entitled "Structured Logic Design of Integrated Circuits Using the Stored Logic Array"and is authored by K. F. Smith, T. M. Carter, and C. E. Hunt. (A complete description of the NMOS PPL cell set is contained in a University of Utah Department of Computer Science internal report entitled "NMOS PPL Cell Set" December 29, 1981.)

The work on the PPL design methodology is divided up into four areas: (1) The verification of the design procedure using the PPL cell set ^{2} by the actual design of selected circuits, (2) The circuit and performance verification of the present PPL cells through actual fabrication of devices to insure that predictable, working circuits result, (3) The development of a logic simulator for simulation of designs using the PPL methodology, and (4) The development of an editor for the specification of PPL designs using graphical input.

The verification of the design procedure using PPLs for circuit design is being done as student projects in two courses (CS539, Fundamental Integrated Circuit Design, and CS572, VLSI Design for Industry.) The PPL is also being used as the building block for circuit objectives in Carter's SICU design system described in Section3.

Designs using PPLs and/or some modified form are also being carried on outside of the University in three different places with somewhat different emphasis. (1) General Instrument Corp. (GI) has committed a major effort for the development of this type of structured logic and has produced a working circuit design which is presently being produced in high volume. GI is working with a set of cells which were developed at the University of Utah about 2 years ago and their efforts are aimed at a production tool using an NMOS process. (2) Boeing Aerospace is working on a development project using CMOS integrated circuits for PPL designs. (3) A group of engineers at Univac in Salt Lake City has begun a project based on the PPL research for design of circuits using high level computer aids. In addition, another group of researchers at Univac in Bluebell are working on this design methodology.

The circuit and performance of the present PPL cells must be verified through actual fabrication and testing. (Support for this work is under the current DARPA contract.) A test mask is being prepared for the fabrication of test circuits containing the entire PPL library of cells. In addition, selected test circuits containing small state machines, generated using the techniques

being developed by Carter Carter, will be fabricated to check out the design technique. Once the masks for these test devices are complete, we will have them fabricated either at the Hedco Laboratory of the University of Utah or at a vendor in Sunnyvale, Calif.

The development of a logic simulator for simulation of designs using the PPL methodology is being funded by Sistemas y Components of Mexico City. Brent Nelson is working on a unique simulator for the PPL design. This simulator is being developed specifically for PPL designs and uses the common data base which is presently the data base in the Computer Vision (CV) CAD system. A design is first done on the CV machine using the PPL methodology and is then transferred to the DECSYSTEM20 where the logic simulation is performed.

The development of an editor for the specification of PPL designs using graphical input is being funded by General Instrument Corp. Dudley Irish is doing a master's thesis on this topic. The editor will be used to do design of the PPL with a Megatek graphics terminal tied to a VAX750 computer. The design process is presently being done on the Computer Vision CAD machine which has some very severe restrictions. These restrictions, as well as many design aids which have been identified over the past two years, will be incorporated into the new editor.

The research work in the area of self-timed systems is a area of activity and is divided into two groups. One group, which is being supported by the DARPA

project is the developing the "one hot" design methodology (Section 3) and is being directed by Lee Hollaar and Tony Carter. The other group, supported by General Instrument Corp., is being directed by Al Davis and Alan Hayes and uses the more conventional four cycle handshake design methodology.

The area of device modelling is a program funded by General Instrument Corp. One objective of the program is to develop techniques for the extraction of SPICE parameters using measurement equipment connected to our department's VAX750 We have purchased and/or have had donated by Hewlett Packard Co. measuring equipment for doing parameter extraction. We will be able to completely characterize the process that we will be using for our circuit fabrication with a complete check of the predicted model. There are two full time VLSI staff members, Charles Hunt and Walter Howard, who are working on this program.

Another objective of the device modelling program is to develop two-dimensional device modelling of short channel, high voltage switching devices using a program called CADDET. This analysis tool performs the solution of the Poisson equation and the current continuity equation in the channel of the device and will serve as an aid in analyzing new device structures. This work is being directed by Kent Smith, Robert Huber, and Robert Bowman and a student, Bob Kent. The work is not directly applicable to the work for DARPA but will serve as a valuable resource for the program.

The last area of VLSI at Utah is the design of switched capacitor filters for the development of a family of structured analog modules which use the PPL methodology for digital control. This program is funded by Sistemas y Componentes (Mexico City.) It involves the actual design and fabrication of switched capacitor filter circuits coupled together with the PPL cells which give the digital control for the analog circuits. The work is being directed by Kent Smith with a full time staff member, Brent Nelson. Here again the efforts are not directly applicable to the work for DARPA but will be a valuable resource.

6 REFERENCES

[1] Reference Manual for the Ada Programming Language, Proposed Standard Document
July, 1980 edition edition, United States Department of Defense, 1980.

[2] Ada to Silicon Project Staff.
Internet Protocol Case Study: Background, and Initial Design.

[3] G. B. Goates; H. M. Waldron III; S. S. Patil; K. F. Smith; and J. A. Tatman.
Storage/Logic Arrays for VHSIC.
In Proceedings of Semi-Custom Integated Circuit Technology Symposium, pages 191-207. Institute for Defense Analysis, Science and Technology Division, May, 1981.
IDA Log No. HQ 81-23646.

[4] M. L. Griss.
A Portable Implementation of Standard LISP.
Utah Symbolic Computation Group Opnote No. 47, University of Utah, August, 1980.

[5] M. L. Griss.
A Portable Standard LISP Programming Environment.
Utah Symbolic Computation Group, Operating Note 53, University of Utah, March, 1981.

[6] M. L. Griss.
Portable Standard LISP Progress Report.
Utah Symbolic Computation Group, Operating Note 57, University of Utah, October, 1981.

[7] M. L. Griss.
Portable Standard LISP: A Brief Overview.
Utah Symbolic Computation Group, Operating Note 58, University of Utah, October, 1981.

[8] Hellestrand, G. R.
MODAL A System for Digital Hardware Description.
In Proceedings, 4th Intl. Symp. on Comp. Hdwr. Descr. Langs., pages 131-137. IEEE, New York, 1979.

[9] Hellestrand, G.R.
MODAL A System for Digital Hardware Description.
In Proceedings, 4th Intl. Symp. on Comp. Hdwr. Descr. Langs.. IEEE, New York, 1979.

[10] L. A. Hollaar.
Direct Implementation of Asynchronous Control Units.
August, 1981.
Submitted to IEEE Transactions on Computers.

[11] Warren Teitelman.
Interlisp Reference Manual.
Xerox-PARC, 3333 Coyote Hill Road,Palo Alto,Calif. 94304, 1975.

[12] Krieg-Bruckner, B., Luckham, D.C.
ANNA: Towards a Language for Annotating Ada Programs.
SIGPLAN NOTICES 15(11):113-122, 1980.

[13] Leung, Clement K.C.
ADL: An Architecture Description Language for Packet Communication Systems.
In Proceedings, 4th Intl. Symp. on Comp. Hdwr. Descr. Langs., pages 6-13. IEEE, New York, 1979.

[14] R. Mathews; J. Newkirk; and P. Eichenberger.
A Target Language for Silicon Compilers.
In Digest of Papers: CompCon Spring 82, pages 349-353. IEEE Computer Society, 1982.

[15] Organick, E. I., and Lindstrom, G.
Mapping high-order language units into VLSI structures.
In Proc. COMPCON 82, pages 15-18. IEEE, Feb., 1982.

[16] S. S. Patil and T. A. Welch.
A Programmable Logic Approach for VLSI.
IEEE Transactions on Computers C-28(9):594-601, September, 1979.

[17] Patil, S. S. and Welch, T.
A Programmable Logic Approach for VLSI.
IEEE Trans. on Computers C-28:594-601, Sept, 1979.

[18] Postel, Jon: editor.
Internet Protocol: DARPA Internet Program, Protocol Specification.
Technical Report RFC 791, Information Sciences Institute, USC, Sept., 1981.

[19] Rattner, J. R.
Functional Extensibility: Making the World Safe for VLSI.
In Kung, Sproull, and Steele (editor), VLSI Systems and Computations, pages 50-51. Computer Science Press, October, 1981.

[20] C. L. Seitz.
System Timing.
Addison-Wesley, Reading MA, 1980, pages 218-262.

[21] H. E. Shrobe.
The Data Path Generator.
In Digest of Papers: CompCon Spring 82, pages 340-344. IEEE Computer Society, 1982.

[22] K F Smith.
Implementation of SLA's in NMOS Technology.
In Proceedings of the VLSI 81 International Conference, Edinburgh, UK, pages 247-256. August, 1981.

[23] K F Smith.
Design of Stored Logic Arrays in I2L.
In IEEE 1981 International Symposium on Circuits and Systems, pages 105-110. April, 1981.

[24] K. F. Smith; T. M. Carter; and C. E. Hunt.
The CMOS SLA and SLA Program Structures.
In H. T. Kung; B. Sproull; and G. Steele (editor), Proceedings of the 1981 CMU Conference on *VLSI Systems and Computatons*, pages 396-407. Computer Science Department, Carnegie-Mellon University, Computer Science Press, October, 1981.

[25] K. F. Smith; T. M. Carter; and C. E. Hunt.
Structured Logic Design of Integrated Circuits Using the Stored Logic Array.
IEEE Journal of Solid-State Circuits tbd(tbd):tbd, April, 1982.

[26] K. F. Smith.
Design of Integrated Circuits with Structured Logic Using the Storage/Logic Array (SLA): Definition and Implementation.
PhD thesis, Department of Computer Science, University of Utah, March, 1982.

[27] Subrahmanyam, P.A.
A Synthesis paradigm for VLSI circuits.
Unpublished, January 1981.

[28] Wile, Dave.
POPART: A Producer of Parsers and Related Tools, System Builder's Manual.
Unpublished, USC/ISI, June 1980.